Module 2: Transaction Management

Author: Parv Bansal
UID – 23BCS13701
Section – Krg 2-A

1. Objective

The objective of the Transaction Management Module is to handle all user financial activities, including income and expense tracking. This module enables users to record, modify, and analyze their financial data efficiently. It ensures that each transaction is securely stored, easily retrievable, and categorized for better financial insights.

2. Features

• Add/Edit/Delete Transactions: Allows users to create, update, or delete financial entries (income or expense) with relevant details such as amount, date, and category.

• Category Management: Users can create and manage custom categories to better organize their financial transactions (e.g., Food, Rent, Salary, etc.).

• Search & Filter Transactions: Users can search transactions based on specific filters like date range, category type, or transaction amount.

• Automatic Balance Updates: Each time a transaction is added, updated, or deleted, the system recalculates and updates the user's balance automatically.

3. Technical Flow

1. User adds a new transaction through the frontend (React form).

2. Axios sends the transaction data to the /transactions API along with the user's JWT token for authentication.

3. Backend validates the token and saves the transaction to the database (MongoDB) with the associated user_id.

4. Redis cache is updated or invalidated to reflect the new data for quick retrieval (monthly summaries, recent transactions).

5. The frontend fetches updated data to refresh the user's dashboard and reflect the new financial summary.

4. Sample Implementation (Node.js / Express)

The following code demonstrates how transactions are created, retrieved, updated, and deleted using Node.js and Express. This implementation also integrates Redis caching to enhance performance when fetching transaction data.

```
// transactionModule.js
import express from "express";
import Transaction from "../models/Transaction.js";
import { authenticate } from "../middleware/auth.js";
import redisClient from "../config/redis.js";

const router = express.Router();

// ============================
// ADD NEW TRANSACTION
// ============================
router.post("/", authenticate, async (req, res) => {
 try {
  const { amount, type, category, date } = req.body;
  const txn = new Transaction({
   user_id: req.user.id,
   amount,
   type,
   category,
   date,
  });
  await txn.save();
  await redisClient.del(`transactions_${req.user.id}`);
  res.status(201).json({ message: "Transaction added", txn });
 } catch (err) {
  res.status(500).json({ error: "Transaction add failed" });
 }
});

// ============================
// FETCH ALL TRANSACTIONS
// ============================
router.get("/", authenticate, async (req, res) => {
 try {
  const txns = await Transaction.find({ user_id: req.user.id }).sort({ date: -
1 });
  res.json(txns);
```

```
  } catch (err) {
    res.status(500).json({ error: "Failed to fetch transactions" });
  }
});

// ============================
// UPDATE TRANSACTION
// ============================
router.put("/:id", authenticate, async (req, res) => {
  try {
    const updated = await Transaction.findByIdAndUpdate(req.params.id,
req.body, { new: true });
    res.json(updated);
  } catch (err) {
    res.status(500).json({ error: "Update failed" });
  }
});

// ============================
// DELETE TRANSACTION
// ============================
router.delete("/:id", authenticate, async (req, res) => {
  try {
    await Transaction.findByIdAndDelete(req.params.id);
    res.json({ message: "Transaction deleted" });
  } catch (err) {
    res.status(500).json({ error: "Delete failed" });
  }
});

export default router;
```

---

5. Detailed Explanation of Code

1. The `/` (POST) route allows users to add a new transaction. The data includes amount, type (income/expense), category, and date. After saving the transaction, the Redis cache for that user's transactions is cleared to maintain consistency.

2. The `/` (GET) route retrieves all transactions associated with the authenticated user, sorted by date in descending order.

3. The `/:id` (PUT) route allows editing or updating a specific transaction using its unique ID.

4. The `/:id` (DELETE) route removes a transaction permanently from the database.

5. All routes are protected by the `authenticate` middleware, ensuring that only logged-in users can access or modify their financial data.

6. Redis Caching Integration

To optimize performance and reduce redundant database queries, Redis is used for caching recent transactions and monthly totals. When a transaction is added, updated, or deleted, the Redis cache is invalidated (`redisClient.del()`) so that the next data fetch retrieves the most recent and accurate information from the database.

7. Data Flow Explanation

1. User interacts with the frontend interface to perform transaction operations.
2. Each operation triggers an Axios API request with the JWT token.
3. Backend validates the token, performs the requested operation (Create, Read, Update, Delete), and responds with JSON data.
4. Redis caching ensures faster access to frequently used data.
5. The frontend updates the dashboard with the new financial data instantly.

8. Summary Table

Process	Description	Tools/Library Used
Add Transaction	Adds new income/expense to database	Express, Mongoose, JWT
Fetch Transactions	Retrieves all user transactions	Mongoose, JWT
Update Transaction	Modifies existing transaction record	Express, MongoDB
Delete Transaction	Removes transaction permanently	Express, MongoDB
Caching	Optimizes repeated queries using Redis	Redis, Node.js

9. Security & Performance Best Practices

- Use JWT authentication to ensure only valid users can access their transactions.
- Invalidate or refresh Redis cache after every database modification to maintain data consistency.

- Validate all input fields (amount, type, category) before saving to prevent data corruption.
- Implement pagination when fetching large sets of transactions to improve performance.
- Encrypt sensitive user data and store minimal information in JWT payloads.
- Perform regular backups of transaction data and maintain audit logs for financial transparency.

10. Conclusion

The Transaction Management Module plays a vital role in handling user financial data by enabling easy tracking, categorization, and management of income and expenses. Through the integration of Redis caching and secure JWT-based authentication, the system ensures both high performance and data security. This module provides the foundation for financial analytics and personalized dashboards in modern web applications.